Table of Contents

About The Complete Friday Q&A: Volume I

Friday Q&A is a biweekly series on Mac programming. It can be found online at http://mikeash.com/pyblog/. Volume I is a full archive of all posts through August 2010.

The author gratefully acknowledges all of the topic and comment contributions to Friday Q&A from its readers.

Mike Ash
mike@mikeash.com
http://mikeash.com/

Introduction

I started NSBlog about five years ago with the intent of just having a place to occasionally write about code and technical matters. Attempting to limit myself to meaty posts ensured that content was sporadic, as meaty topics were hard to come by.

Eventually these posts drew in a relatively substantial amount of traffic. I wanted to satisfy this newfound readership, but thinking of interesting things to write about was tough. The solution to this dilemma was lurking in the cause: pull topics from the readership!

Thus I announced Friday Q&A. Each week I would take a reader-submitted topic and use that for a blog post. I didn't have high expectations but it seemed like it was worth a shot.

Response was tremendous! Each post gets a sizeable amount of readers who have posted many intelligent and interesting comments. And although there have been a couple of pauses in the schedule (and a shift to a biweekly schedule once the weekly schedule became too demanding), Friday Q&A has been more or less continuously published ever since. I have been honored to see a large number of unjustifiably kind comments and recommendations about the series.

The idea for a book version has been floating around for quite a while. After a disastrous encounter with the publishing industry, the thought of self-publishing began to look more attractive. With the introduction of the iPad and the success of iBooks, selling as an ePub through Apple's store seemed like the logical course of action.

This book is a compendium of all Friday Q&A articles from the first one on December 19, 2008 through the latest one as of this writing on August 27, 2010.

The content is mostly unedited from the original posts. This means that each chapter still talks about "last week", and encourages the reader to return "next week". There are two reasons for this. First, Friday Q&A is an inherently temporal series and this preserves that feel in the book. Second, fixing up all of the articles to appear as though part of a reference book would have been a lot of work, and like most programmers I am lazy.

Some articles link to code in my public Subversion repository. All of these links remain valid. However, you may wish to check my github page first, as I have moved several projects there and some of them are more up to date than the Subversion copy.

It's been a pleasure to write Friday Q&A over these past two years, and I hope to continue for many more. Although the "next week" is out of date, the reader-driven nature of the series is not. So as always, if you have an idea for a topic that you'd like to see covered in Friday Q&A, send it in!

Acknowledgements

I would like to thank my reviewers, whose valuable input dramatically improved this book. They are: Steven Vandeweghe, Vadim Shpakovski, Matthias Neeracher, Phil Holland, Landon Fuller, David Helms, Joshua Pokotilow, Mike Shields, Jeff Schilling, Jordan Breeding, Hamish Allan, Eimantas Vaiciunas, Ilya Kulakov, Cédric Luthi, Alex Blewitt, and Kevin Avila.

I would also like to thank everyone who contributed the topic ideas used throughout this book. Their names can be found at the beginning of each chapter.

Finally, I would like to thank everyone who has commented on one of my posts, e-mailed about Friday Q&A, or simply read it. No matter what your contribution, it is appreciated.

Friday Q&A 2008-12-19: Multithreaded Programming and Parallelism Overview

Related Articles

Great response last week. This week I'm going to merge Sam McDonald's question about how I got into doing multithreaded programming and Phil Holland's idea of talking about the different sorts of parallelism available.

Like a lot of computer programmers, I was always interested in making code run fast. This led to better languages (I started in BASIC!), micro-optimization, and algorithms, but ultimate performance means multiprocessing. The distributed.net and SETI@Home projects showed the power of distributed computation.

Multithreading was also interesting in coming to OS X from the old Mac OS, where multithreading was a lot more limited and difficult. At the time it wasn't about performance, since most machines had only one CPU. But multithreading has lots of other benefits for organization, design, and interactive GUIs so it was still highly useful.

Then the ongoing multicore revolution kicked off and made it clear that multithreading was the way to go.

That's the why. The how is pretty boring. Just lots of work, reading, and experimentation on all sorts of multiprocessing, not just threading. They're very different, but many concepts are the same, and ideas from one can often help with the others. As with most things, practice and experience makes a big difference.

So then we have the different forms of parallel processing available. There are actually a lot of these, and I'm probably doomed to miss some, but:

1. **Distributed computing.** Probably the most visible example of this one for Mac developers is distributed builds in Xcode. This is generally the most difficult to build, the most expensive to take advantage of for the user, and therefore the least useful. Bandwidth and latency between computers are horrendous compared to what you get within a single machine, so it's hard to write something that goes fast. Xcode can get away with it because it (usually) does a lot of processing for each bit of data that it processes. Beyond the difficulty of achieving speed, you also have to deal with a much more error-prone environment. You want to recover gracefully if the user wanders out of wifi range, not lose a bunch of data. Most of the time this is not worth it, especially if you're going to be shipping consumer-level software.
2. **GPGPU.** Basically using the video card for computation. This is what GPULife does. It's capable of immense power. A top-of-the-line video card can easily outperform a top-of-the-line CPU by a factor of 50 with the right program. It's also really hard. GPUs are extremely parallelized and have a considerably different architecture from CPUs, so coding for them is hard and making them go fast is harder. (Although even slow GPU code can run really fast due to the amount of power available.) Technologies like CUDA and OpenCL promise to make this sort of thing a lot better, although you're always going to be dealing with the fact that it's a massively parallel system with really different performance characteristics. My recommendation here is to wait for Snow Leopard and then hope OpenCL delivers on its promise.
3. **Multiple processes.** Again Xcode is a prominent example of this approach, where you can see it spawn multiple instances of gcc when compiling. This is often talked about as being an easier, safer way to go than multithreading because the OS protects processes from each other and forces a better separation of concerns. I don't buy it, personally. For just about any non-trivial program, a dead subprocess is going to mean that the whole thing comes crashing down, and all you've done by splitting it into multiple processes is make it harder to debug. What's worse, OS limits on the number of processes tend to be frighteningly low, so your program would need to gracefully handle being unable to spawn as many subprocesses as it likes. (And all the other apps on the system would need to as well!)
4. **Multithreading.** The standard technique. Often very difficult to get right, and very difficult to debug, but potentially very rewarding in terms of performance. Threads can also help to better organize a program. It's often much cleaner to put long-running processing or blocking operations into a separate thread than to try to multiplex them together.

Multithreading is the one we're most familiar with and the one that's the most generally useful. It's useful because it's very generalized, so you have various ways to use multithreading to actually get things done:

1. **Locks.** "Standard" multithreading. You have shared data protected by locks. Acquire the locks before you fiddle with the data. Often used to build more sophisticated machinery. This level can be tricky to get right so I recommend avoiding it where you can, and using it sparingly to build better abstractions where you must.
2. **Message passing.** With message passing, you avoid shared data, and have threads communicate using message queues instead. (The message queues generally have shared data inside them, but that's an implementation detail.) Cocoa has some nice facilities for this with the `-performSelectorOnMainThread:...` and `performSelector:onThread:...` calls. The threading-heavy language Erlang uses this model extensively and is the main force behind its multithreading power.
3. **Operation units.** This is kind of like message passing, except the operations just fly off and get executed on a queue which uses threads outside your view. When set to only execute one operation at a time, a queue can act like a synchronization point, replacing locks in a way that's often easier to work with. NSOperationQueue provides this and Grand Central Dispatch in Snow Leopard is rumored to provide similar facilities.
4. **Atomic/transactional objects.** Rather than using mutual exclusion (locks, queues) to avoid destroying shared data, you can build objects that operate using transactions. Grab a snapshot, make changes, commit them. (Often this is implemented as a loop: snapshot, change, try to commit and start over with a new snapshot if something changed in the middle.) TransactionKit is a great example of this sort of thing in a Cocoa context.

As for what to use, here are my thoughts. Avoid distributed computing unless your code is going to be run by a single client with a lot of available hardware. Being able to snarf up CPU cycles from idle hardware sitting around in the user's house sounds cool but just doesn't pay off most of the time. Avoid GPGPU on the Mac until Snow Leopard ships unless you have a *really* good application for it. OpenCL will make GPGPU a lot more practical and flexible, so trying to shoehorn your computationally expensive code into GLSL or CoreImage today just doesn't seem worth it.

Using multiple processes is a good idea if the subprograms are already written. If you're invoking gcc as a subprocess, invoking it simultaneously on four files instead of one by one is pretty easy. If you're writing your code from scratch, I don't recommend it unless you have another good reason to write subprocesses, as it's difficult and the reward just isn't there.

For multithreading, concentrate on message passing and operations. Multithreading is never easy, but these help greatly to make it simpler and less error prone. Good OO design will also help a lot here. It's vastly easier to multithread an app which has already been decomposed into simple objects with well-defined interfaces and loose coupling between them.

Friday Q&A 2008-12-26: Blocks in Objective-C

Related Articles

Welcome to another Friday Q&A. This week I thought I would take fellow amoeboid Jeff Johnson's suggestion and talk about blocks in Objective-C.

The word "blocks" is kind of ambiguous, so to clarify, I'm not talking about the compound statement structure which has existed in C since the beginning of time. I'm talking about a new addition to the language being created by Apple which adds anonymous functions to the language. [Note: since this chapter was written, Apple's blocks implementation has been made public and is now completely mature and usable. While this chapter remains relevant, good up-to-date documentation on blocks can now be found on developer.apple.com.]

Since they're not available to the public in finished form yet, the discussion is going to be a bit imprecise in terms of syntax. But since I mainly want to talk about what they will do for us and not the absolute precise details of how to type them out, that's not a big problem. First let's see how they look:

```
x = ^{ printf("hello world\n"); }
```

That's a block. The funny caret before the braces is what distinguishes it from boring old compound statements. Now we can simply call this block like so:

```
x();
```

And the resulting code will print "hello world". Now let's introduce a couple of parameters:

```
x = ^(int a, char *b){ printf("a is %d and b is %s", a,
b); }
```

And then we can call this just the way you'd think:

```
x(42, "fork!");
```

Now let's remove the parameters again:

```
int a = 42;
char *b = "fork!";
x = ^{ printf("a is %d and b is %s", a, b); }
x();
```

This illustrates one of the really interesting things about blocks: they can capture variables from their enclosing scope. This is not particularly interesting here (why didn't we just pass a and b when invoking it?) but it gets really interesting when we start passing the block around to other functions:

```
int a = 42;
char *b = "fork!";
callblock(^{ printf("a is %d and b is %s", a, b); });
```

When the `callblock()` function calls that block, the block will still get access to our local variables a and b even though we never passed them to the function explicitly.

We're just about done with the basics of what blocks are. One more quick example, a block that returns a value:

```
x = ^(int n){ return n + 1; };
printf("%d\n", x(2));
```

This code will print "3". Note that there is no need to declare the type of the return value as the compiler can simply infer it from the return statement.

So what's the big deal? A major advantage of blocks is that they essentially allow you to write your own control structures in the language without having to alter the compiler. As one example, take the `for(... in ...)` syntax that appeared in Leopard. This syntax is a wonderful addition to the language. Previously we had to write a bunch of code just to iterate over an array:

```
NSEnumerator *enumerator = [array objectEnumerator];
id obj;
while((obj = [enumerator nextObject]))
    // finally we can do something with obj
```

And the new syntax cuts this down to a single line:

```
for(id obj in array)
```

Which is great. The only trouble is that we went years and years without it. We had to wait for Apple to add it for us. With blocks, no more! You don't get quite the same syntax, but you can get the same convenience with a method you wrote entirely yourself:

```
my_for(array, ^(id obj){ /* loop body goes here */ });
```

Or in a perhaps slightly stranger but much more interesting object-oriented form:

```
[array do:^(id obj){ /* loop body goes here */ }];
```

The implementation of the `-do:` method is left up to the reader, but rest assured that it's relatively simple.

As another example, consider the `@synchronized` directive. This could be redone using blocks too:

```
[obj synchronized:^{ /* this is protected by the lock */
}];
```

OK, you say, I get it, but what's the big deal? After all, `for/in` and `@synchronized` are already part of the language, why would you rewrite them?

Of course you wouldn't. That would be silly. Those examples serve only to illustrate the idea: that you can build your own control structures. But of course it's only interesting to build control structures that are new! So here are some ideas.

- Open a file and ensure that it gets closed when you're done:

  ```
  [[NSFileHandle fileHandleForReadingAtPath:path]
    closeWhenDone:^(NSFileHandle *handle){
      /* use handle here */
  }];
  ```

- Build a new array by working with the objects of an existing one:

  ```
  newArray = [existingArray map:^(id obj){ return [obj
  stringByAppendingString:@"suffix"]; }];
  ```

- Filter the contents of an array:

  ```
  newArray = [existingArray filter:^(id obj){ return
  [obj hasPrefix:@"my"]; }];
  ```

- Main thread synchronization:

```
/* threaded code */
PerformOnMainThread(^{ /* synchronized code */ });
/* more threaded code */
```

- Delayed execution:

```
PerformWithDelay(5.0, ^{ /* will run 5 seconds later
*/ });
```

- Parallel enumeration:

```
[array doParallelized:^(id obj){
    /* will get executed on all of your CPU cores at
once */
}];
```

And many other examples abound.

Another place where blocks will make things much nicer is when dealing with callbacks. If you've ever written much Cocoa code you've probably had to write a sheet callback, and it's a pain in the ass. If you need to pass variables through to the other side then it gets really frustrating with code like this:

```
- (void)method {
    int foo;
    NSString *bar;
    /* do some work with those variables */
    NSDictionary *ctx = [[NSDictionary alloc]
initWithObjectsAndKeys:
        [NSNumber numberWithInt:foo], @"foo",
        bar, @"bar",
        nil];
    [NSApp beginSheet:sheet
        modalForWindow:window
        modalDelegate:self

didEndSelector:@selector(methodSheetDidEnd:returnCode:contextInfo:)
        contextInfo:ctx];
}

- (void)methodSheetDidEnd:(NSWindow *)sheet
returnCode:(int)code contextInfo:(void *)ctx {
    NSDictionary *ctxDict = ctx;
    [ctxDict autorelease];

    int foo = [[ctxDict objectforKey:@"foo"] intValue];
    NSString *bar = [ctxDict objectForKey:@"bar"];
    /* do some more stuff with those variables */
}
```

Wow! What a pain that is. Since I removed all the stuff that does work, nearly everything that remains is just boilerplate. Horrible boilerplate whose only purpose is to tell the sheet who to call, and to pack up local information in a way that the sheet can give it back to you later on. Now let's imagine we were redoing this API using blocks and see how it would look:

```
- (void)method {
    int foo;
    NSString *bar;
    /* do some work with those variables */
    [sheet beginSheetModalForWindow:window
didEndBlock:^(int code){
        /* do stuff with foo */
        /* do stuff with bar */
        /* do stuff with code, or sheet, or window, or
anything */
    }];
}
```

Isn't that great? All that horrible boilerplate just flies right out the window. Code flow suddenly becomes completely logical, you can read it top to bottom, and you can access any local variables you please.

Let's take another example, sorting an array with a custom comparison function using some variables that you pass in. NSArray has functionality for this, with the `-sortedArrayUsingFunction:context:` method. The old-style code is annoying, and I'm not going to write it. It's much like the sheet method above. You have to define a separate function, way outside of your code where it's not really visible. You have to set up the context to pass into it. If you're passing more than one thing then you have to pass a dictionary (and unpack it) or a pointer to a struct. Now here's the blocks version of a custom comparator:

```
sorted = [array sortedArrayUsingBlock:^(id a, id b){
    /* compare, use local variables to decide what to do,
run wild */
}];
```

And that's all there is to it.

Callbacks are one of the most powerful things in C and Objective-C but in many situations their use can be extremely difficult and unnatural. Blocks promise to allow callbacks and custom control constructs to be created and used in a much more natural fashion.

So far I've only shown examples of using a blocks API, but how about creating one? Well, it's a little worse, but not much. The only problematic thing is that the syntax for declaring a block type is kind of ugly, as it's modeled after function pointer syntax. But it's not too bad, and the rest is nice and simple. For example, here's how you could write that `-map:` method from above:

```
- (NSArray *)map:(id (^)(id))block { // takes an id,
returns an id
    NSMutableArray *ret = [NSMutableArray array];
    for(id obj in self)
        [ret addObject:block(obj)];
    return ret;
}
```

Pretty straightforward, especially considering the power it gives us.

Information on Apple's implementation of blocks is still a bit sparse. Some more details can be found in a mailing list post to the Clang development list. For more purely conceptual ideas on how blocks can be used, check out the Smalltalk language, where blocks are used for virtually every control structure right down to if/then and basic loops.

Here's hoping that blocks allow for some major changes in how we work on Snow Leopard!

Friday Q&A 2009-01-02: Pros and Cons of Private APIs

It's a new year, and that means a new Friday Q&A! This week I'm going to take Steven Degutis's suggestion and discuss the ups and downs of using private APIs.

Getting Started

I'm not going to discuss what private APIs are out there or how to figure them out, as that would cause me to badly miss my deadline and make this thing way too long. Instead I just want to address this question: *should* you use them at all, and if so, when?

There are two pretty obvious extremes to the answer, and a lot of people who believe each end. One extreme is that private APIs should never be used, period, full stop. They're bad, don't want to touch them, don't even acknowledge that they exist. The other extreme is that they're fine and dandy, use them like you'd use anything else.

As with most things, I believe the truth lies somewhere in the middle. But where, exactly, and how do you determine if something is worth using?

First let's review the disadvantages, which you're probably familiar with already. An API is essentially a contract between the creator of the API (generally Apple in the context of this blog) and the user of that API. When an API is public, the creator promises not to change that API in an incompatible fashion. With private APIs no such promise exists, and they can change at any time. This change can cause your application to malfunction, crash, or refuse to start.

And the advantages? Well that one's easy. Private APIs let you do stuff you couldn't otherwise do.

So like most of engineering, it's a tradeoff. You have benefits and disadvantages, and you have to decide which one is more significant.

Elements of the Tradeoff

Using a private API is, ultimately, a maintenance issue. (Except on the App Store, where it's a legal issue, but that's outside the scope of this post.) If you use nothing but public APIs, your app is basically guaranteed to work forever. (Where "forever" really means "until Apple decides not to maintain backwards compatibility anymore". But note that ancient PowerPC-only Carbon apps still run on the latest Mac OS X, and that Classic didn't disappear until 10.4; Apple still keeps old stuff working for a good long time.) If you use a private API, your app is likely to break at some point.

But when? That's one of the big questions you need to answer. There are basically four levels to consider:

1. **Never.** Sometimes a private API may be so fundamental and so widely used that it gets essentially fixed in stone despite not being public. A good example of this on Mac OS X is the `mach` APIs, which are technically private but which underly everything at a very fundamental level.
2. **Major releases.** Most private APIs fall into this category, where you can be reasonably (although never 100%) confident that they will continue to work throughout the lifetime of the current major OS release. In other words, it will keep working on 10.5 but is likely to break on 10.6. Typically private APIs end up forming part of a support structure for the public APIs and can't be changed without a major reworking of those public APIs, and that only happens with a new major release.
3. **Minor releases.** Occasionally something can't even be relied upon to keep working during the life of a major release. Early versions of LiveDictionary were like this. They relied on fiddly internal details of WebCore's implementation, like C++ method and ivar layout. These offsets were subject to change at pretty much any time, so LiveDictionary generally broke every single time Safari got updated. (Later on, public APIs became available that I was able to use instead, which solved the problem once and for all. For more details of what was going on under the hood in those dark days before the public APIs were available, see Hacking C++ From C.)
4. **Any time.** Generally this means that you aren't using the private API right. This is much more common than with public APIs since you don't have any documentation, you have no guarantee as to the API's requirements, constraints, preconditions, postconditions, etc.

Another big question you need to answer is how bad the break, when it comes, is likely to be. Again, there are different levels to consider:

1. **No effect.** It's unlikely that you'll get here. If there's no effect from having it break, why are you even using the thing?
2. **Lose a feature.** Often you can write your code in such a way that the breakage is *likely* to be detectable and so you can simply disable whatever feature uses it.
3. **Crash your app.** This is pretty common.
4. **Crash other apps.** For developers of stuff that loads into other programs this is very common, for self-contained processes not so much. LiveDictionary did this. When LiveDictionary broke, it didn't just crash, it crashed Safari too.

Which category you fall into depends not only on what you're using but how. For example, LiveDictionary would toss up a warning and offer to disable itself for the duration if the WebCore version was higher than what it knew about. A brave user could try to use it anyway (and there was at least one time when that version changed in unexpected ways and stopped this precaution from working) but this helped a lot. Obviously the higher up this list you are, the better off you are.

And lastly you need to figure out how long it will take you to fix the break. This depends greatly on your skill, your availability, what you're using, how critical it is, how it broke, and other such factors.

Coming to a Conclusion

Now you have enough information to run the cost/benefit analysis. The benefit side is pretty easy. The cost side can be determined by looking at how often you're likely to break, how bad it's likely to be, and how much time and effort it will take to fix. If it's a huge feature and will almost never break and will be trivial to fix when it does, then go for it. If it's a minor feature and will cause huge problems when it breaks every three months, pass. For LiveDictionary, the entire app was built around this feature, so it was worth it even though it required frequent difficult fixes.

Remember that the cost is not just to you, but to your users. If you're really unlucky, the break will be so bad that it's not even obvious that it's your fault, and they'll figure it out only after much head-scratching. Once they do figure it out, they will hate you if your fix doesn't come really fast. This means that for a really crucial and breakable feature, you need to stay available and ready to create and release a fix.

Private APIs can be invaluable, but their use must be weighed carefully. Sometimes it pays off very well, and sometimes it's a terrible choice. By carefully examining your app's vulnerability to breakage and your ability to fix it, you can decide whether it's the right move for you.

Friday Q&A 2009-01-09: Thread Safety in OS X System Frameworks

Related Articles

Greetings one and all. I caught my mistaken writing of "2008" in this blog post title almost instantly instead of only noticing after I'd already posted it like I did last week, so the year must be coming along. Welcome to the second Friday Q&A of 2009 (and only the fourth in all human history!) where I'll be taking Ed Wynne's suggestion and talking about the various meanings and implications of thread safety as they apply to Mac OS X system frameworks.

Thread Safety? What's That?

Hopefully not too many readers are actually asking the above questions, but just as a quick refresher, thread safety is about whether it's safe to access a particular module, API, or data structure from multiple threads. These things are typically unsafe due to making assumptions of single-threadedness, such as updating multiple pieces of data in a non-atomic fashion, in such a way as to expose inconsistent data to the outside world. There's the classic example:

```
x++;
```

Which is not thread safe (assuming `x` is globally accessible) because down at the very bottom it breaks down into multiple operations:

```
get x
increment
store x
```

And if multiple threads are doing this at once, they interleave and you miss increments. Not too dire here, but apply it to pointers and objects and you can hopefully see why you'll crash at best, and silently corrupt data if you're unlucky.

So to start off with there are two kinds of thread safety in the world:

1. **Not thread safe.** The normal state. Code is not thread safe by default. Special effort needs to be taken to make it thread safe, and if you haven't done it, your code falls into this category.
2. **Thread safe.** Can be called from any thread without a care or worry. Nice to have, often painful to make.

Three Kinds

But what does this really *mean*? Well, **thread safe** is easy enough to understand. But **not thread safe** can't really mean it can't be called from any thread, because all code runs from *some* thread.

Of course what it really means is that this code can't be run from *more than one thread at the same time*.

But that doesn't really do it either. For example, `NSMutableArray` is not thread safe. But you can call NSMutableArray from multiple threads simultaneously, as long as each thread is working on a different array. So maybe we should say that thread unsafe means that the code can't be run *on the same data* from more than one thread at the same time.

Well, that's better, but still not there. Take the `atoi()` function. Not thread safe, says so in the man page. But you only ever feed it constant data, and it's unsafe even if you feed it completely different data on your different threads. What's the deal? Simple: behind the scenes, it has some shared data.

How can you tell one from the other? We'll need another classification:

1. **Never thread safe.** The normal state. Code is not thread safe by default. Special effort needs to be taken to make it thread safe, and if you haven't done it, your code falls into this category.
2. **Not thread safe with shared data.** Can safely be called from multiple threads simultaneously as long as each thread is dealing with a distinct set of data.
3. **Thread safe.** Can be called from any thread without a care or worry. Nice to have, often painful to make.

It's actually really easy to write code that falls into category #2. All you have to do is not have any global state, which is pretty common anyway. If you're writing an array class,

your method for adding a new object to the array isn't going to deal with global state, it's going to deal with that one array. So while #1 may be the "normal state", #2 is actually really easy to come by, and most code falls into that category.

The System Screws It All Up

These categories are sufficient in a relatively simplistic program which controls every action taking place and for which all the code is known. It gets more complicated when you start pulling in a ton of big, complex external frameworks such as AppKit and Foundation. Take `NSView` as an example. It can fall into category #1 or #3 depending on what you're doing with it. (Drawing is safe, creation/resizing/etc. is unsafe.) But that #1 is complicated by the fact that the shared global data which makes NSView unsafe can be *accessed by code that isn't yours*.

NSView isn't just unsafe from multiple threads, it's **main thread only**. This is because your NSView doesn't just belong to you, it belongs to the framework. And this means that you can't synchronize all accesses to it, because some of those accesses come from code that does not belong to you! Let's put this in its own paragraph, because it's important:

If an API is never thread safe and you do not absolutely control every access to this API, then you can only call it from the main thread.

And since virtually every system API is going to be, at least potentially, called by other system APIs, we can rewrite our three types of thread safety:

1. **Main thread only.** The normal state. Code is not thread safe by default. Special effort needs to be taken to make it thread safe, and if you haven't done it, your code falls into this category.
2. **Not thread safe.** Can safely be called from multiple threads simultaneously as long as each thread is dealing with a distinct set of data.
3. **Thread safe.** Can be called from any thread without a care or worry. Nice to have, often painful to make.

Singletons

Keep in mind that singletons qualify as global shared data. This has an important impact on their thread safety. Practically speaking, it means that singletons provided by system frameworks only ever fall into category #1 or #3. Take NSFileManager as an example. It's listed as not being thread safe. What this really means is that `[NSFileManager defaultManager]` can only be safely used from the main thread, because you can't control what other code might access it. (On 10.5 and above you can `alloc/init` your own private instances which then fall into category #2.)

Terminology and the Apple Way

This is all fine and dandy except that Apple, in their infinite wisdom, does not always distinguish between **main thread only** and **not thread safe**. To make things worse, they even sometimes use the term **thread safe** to mean what we have defined here as **not thread safe**.

Let's take that second one first, because it's pretty weird. As a concrete example, look at the `CFNetDiagnostics` API. The documentation for this API is full of quotes like this:

> `This function is thread safe as long as another thread does not alter the same CFNetDiagnosticRef at the same time.`

Huh??

So why is it labeled "thread safe"? What they're trying to convey here, through the fog of inadequate terminology, is that this API falls into category #2 and not category #1. In other words, you can use it from any thread as long as only one thread at a time is using this API on any given piece of data. This as opposed to an API which requires you to call it only from the main thread.

Other APIs are less explicit about it. The `Search Kit` reference simply states "Search Kit is thread-safe". And yet I'm pretty sure it's not. Again, it's trying to convey that Search Kit is in category #2 rather than category #1.

Why do they do this? Well, back in the day, on the classic Mac OS, nearly all code ran in what might be considered the "main thread" today. As a consequence, nearly every API required only calling it from there. Being able to run from multiple threads was novel and unusual and was worth documenting. Alas, not only does this no longer make sense on Mac OS X, but this sort of terminology abuse is actively destructive because it ends up making guarantees which aren't actually true.

As an example of the first, look at `NSAppleScript`. In the big master guide it's marked as being not thread safe. This is true! However what they don't tell you is that `NSAppleScript` can only be safely used from the main thread, due to AppleScript itself being a main thread only API. And yet it's right next to other classes such as `NSMutableArray` which are clearly category #2.

Figuring It Out
So we've established the three basic categories of thread safety, and we've established that Apple doesn't consistently distinguish between them in its documentation. So what do we do?

Fortunately it's *usually* possible to figure out the real story.

1. Check the documentation. Not only the API documentation but also the big list. Is the API listed as being thread safe? Is it written in a relatively unambiguous way that makes it clear that this really is thread safe, and not the "thread safe" that means "not thread safe"? Fortunately for us, this abuse of the term "thread safe" is relatively rare and relatively obvious. It generally shows up in older APIs

which have no reason to be thread safe in the first place. If after all this you have determined that your API is **thread safe** then you're done! If not go to the next step.

2. When in doubt, assume it's unsafe. In the absence of an explicit guarantee of thread safety, consider the API to be unsafe. But what kind of unsafe? That's the tricky thing.
3. Does it access shared global data? Singletons fall into this category, as do things like user interface elements. If the answer is yes, then it's category #1: **main thread only**.
4. Does it potentially invoke other code which may not be thread safe? `NSAppleScript` falls into this category (scripting additions) as do things like user interface elements which may broadcast notifications when they're manipulated. Again, if the answer is yes, it's main thread only.
5. If you got this far then it's *probably* category #2, **thread unsafe**, and usable on any thread as long as you synchronize calls on shared data.
6. To verify, think about how simple or self-contained an API is. If it's pretty self-contained then it's very likely category #2. If it calls out to a zillion other things then it's very likely category #1. Therefore we can be pretty sure that NSMutableArray (self-contained) is merely not thread safe, whereas NSAppleScript (calls out to all sorts of other stuff, including arbitrary third-party components) needs to run on the main thread.

It would be good if Apple would properly distinguish between the different kinds of thread safety. However you can *usually* do a good job of figuring out any given API if you work it through.

Friday Q&A 2009-01-16: Interprocess Communication

Related Articles

Happy Friday to everyone, and welcome back to another Friday Q&A. This week I'll be taking Eren Halici's suggestion to discuss the various ways to do interprocess communication on OS X.

IPC is an interesting and sometimes complicated topic, especially on OS X, which has a veritable zoo of IPC techniques. It can be hard to decide which one to use, and sometimes hard to even know what's available.

OS X is a funny mixture of mach and UNIX so you end up with IPC mechanisms from both:

- **Mach ports:** The fundamental IPC mechanism in mach. Fast, light-weight, extremely capable, and difficult to use. Mach ports will not only let you talk to other processes, but do things as drastic as inject code into other people's programs. The poor state of the mach documentation makes it hard to get started and easy to make mistakes with it. On the other hand, the core `mach_msg` function is probably the most optimized syscall in the system, so they're really fast to use, and your machine will barely blink if you decide to allocate a million mach ports at once.
    - **CFMachPort:** A very thin wrapper around mach ports. `CFMachPort` essentially exists to allow a mach port to be used as a runloop source. It can also help with creating and destroying the ports. It helps a little with receiving messages and not at all with sending them.
    - **CFMessagePort:** This nice CoreFoundation wrapper around some mach functionality makes it easy to set up synchronous back-and-forth communication between two unrelated processes. You can start a server with just a few lines of code. Another program can then look up that server by name and message it. You get the speed advantages of mach without all the messy stuff going on underneath.
    - **NSPort/NSMachPort/NSMessagePort:** Cocoa has some mach port wrappers too. They're mainly geared toward use with Distributed Objects (more on that below) but can be used on their own as well, if you're brave.
- **POSIX file descriptors:** There are actually several kinds of these but they can all be used with the typical `read` and `write` calls once they're set up.
    - **Pipes:** The archetypal POSIX IPC mechanism. If you've ever used the `|` pipe operator in a UNIX shell, you've used a pipe. Pipes get created in

pairs within the same process, so they're good for communicating between parents and children (or between two children of a single, coordinating parent) but not so good for communicating between unrelated processes. Make them with the `pipe` call.

    - **FIFOs:** It's like a file, but it's like a pipe! A FIFO gets an entry in your filesystem, just like a file, but writes don't go to the filesystem, instead they go to whatever process has opened the fifo for reading. You can make these with the `mkfifo` call. The end result is a pipe that has a filesystem entry, which can make it easy for two unrelated processes to hook up. The processes don't even have to know that they're talking to a fifo. Try it out in your shell:

```
$ mkfifo /tmp/fifo
$ cat /tmp/fifo

# in another shell
cat > /tmp/fifo
type some junk here
```

    - **Sockets:** You probably know these from working with TCP/IP, but they can also be used to communicate locally, and not just by connecting to `localhost`. If you create a socket in the `AF_UNIX` family you get a socket that's only for local communication and uses more flexible addressing than TCP/IP allows. `AF_UNIX` sockets can be created using a filesystem path much like a FIFO by using the `socket` and `bind` calls, but allowing multiple clients and more options for how the communication works. They can also be created anonymously using the `socketpair` call, giving you something much like a pipe, except bidirectional.

- **Shared memory:** Shared memory is a magical piece of memory which appears in multiple processes at once. In other words, you write to it from process A, and read from it in process B, or vice versa. This tends to be very fast, as the data itself never touches the kernel and doesn't have to be copied around. The downside is that it's really difficult to coordinate changes to the shared memory area. You essentially get all of the disadvantages of threaded programming *and* most of the disadvantages of multi-process programming bundled together in one neat package. Shared memory can be created using either mach or POSIX APIs.
- **Miscellaneous, not really IPC:** There are some techniques which don't really count as "IPC" but can be used to communicate between programs if you want to.
    - **ptrace:** This system call exists mainly for writing debuggers, but could in theory be used to do non-debugger things too. Not recommended, included only for completeness.

    - **Files:** Sometimes it can be useful to communicate using plain old files. This can be as simple as creating a lock file (a plain empty file that works simply by being there) for mutual exclusion, or you can transfer actual data around by writing it to a file, then having the other program read it. This tends to be inefficient since you're actually writing to the filesystem, but it's also easy and nearly universal; every application can read files!

Those are all what I would call system-level functionality, things which are either provided directly by the kernel/libSystem, or which are thin wrappers around them. OS X also provides a bunch of higher-level IPC mechanisms at the framework level:

- **Apple Events:** Scourge of the Skies, Champion of the Ugly Contest, King Slow, Emperor Horrible. Apple Events are all of these things, but they're also tremendously useful. They're the only IPC mechanism which is universally supported by GUI applications on Mac OS X for remote control. Want to tell another application to open a file? Time for Apple Events. Want to tell another application to quit gracefully? Apple Events time. Underneath it all, Apple Events are built on mach ports but this is mostly not exposed in the API.
    - **AppleScript:** Everything Apple Events is and worse, but still often useful, AppleScript is a scripting language built on top of Apple Events. Generally it's best to avoid AppleScript and simply send the corresponding raw Apple Events instead, either directly or through a mechanism like Scripting Bridge. AppleScript support is the standard way to allow users to script your application, although if you ever try to add AppleScript support to your application you'll find yourself wishing for a different standard.
- **Distributed Objects:** It's like Objective-C, but it happens over there! DO gives you proxy objects that can be used (mostly) just like local objects, with the exact same syntax and everything, except that your messages fly across to the other process and get executed there. DO normally runs over mach ports but can also be used with sockets, allowing it to work between computers as well. DO is really cool technology and it's the sort of thing that tends to blow people's minds when they come to Objective-C from lesser languages such as Java or C++. Unfortunately DO is also really old and crufty and tends to be strangely unreliable. This is especially true when using it with sockets to talk to remote machines, but is even true when using it locally. DO is also completely non-modular, making it essentially impossible to swap out the IPC mechanism it uses for something custom (like if you want to encrypt the stream). It is worthy of investigation if only to learn about how it works, and despite the shortcomings can still be very useful in certain situations.
- **Distributed Notifications:** These are simple one-way messages that essentially get broadcast out to any process in the session that's listening for them. Extremely easy to use, and available in both Cocoa and CoreFoundation flavors. (And they interoperate!) The downside is that they don't guarantee delivery and

they're very resource-intensive due to potentially messaging every application on your system. They would be completely unsuitable for something like transmitting a large picture to another process, but are great for simple one-off things like "I just changed my preferences, re-read them now". Internally this is implemented by using mach ports to talk to a centralized notification server which manages the task of getting notifications to where they want to go.

- **Pasteboard:** Probably the IPC mechanism that you've directly used the most. Every time you copy and paste something between applications, that's IPC happening! Inter-app drag and drop also uses the pasteboard, and it's possible to create custom pasteboards for passing data back and forth between applications. Like distributed notifications, pasteboards work by talking to a central pasteboard server using mach ports.

So which one is right for you? Well, it all depends on what you're doing. I've used nearly every one of these to accomplish different things over the years. You'll have to see which one fits your problem best, and I hope the above gives you a good place to get started.

Friday Q&A 2009-01-23: How Key-Value Observing Works

Welcome to the first Friday Q&A of the new Presidential administration. Unlike Mr. Obama, I'm afraid of change and so this week's edition will be just like all the other ones. This week I'll be taking Jonathan Mitchell's suggestion to talk about how Key-Value Observing (KVO) is actually implemented at the runtime level.

What Is It?

Most readers probably know this already, but just for a quick recap: KVO is the technology that underlies Cocoa Bindings, and it provides a way for objects to get notified when the properties of other objects are changed. One object *observes* a key of another object. When the observed object changes the value of that key, the observer gets notified. Pretty straightforward, right? The tricky part is that KVO operates with no code needed in the object being observed... usually.

Overview

So how does that work, not needing any code in the observed object? Well it all happens through the power of the Objective-C runtime. When you observe an object of a particular class for the first time, the KVO infrastructure creates a brand new class at runtime that subclasses your class. In that new class, it overrides the set methods for any observed keys. It then switches out the `isa` pointer of your object (the pointer that tells the Objective-C runtime what kind of object a particular blob of memory actually is) so that your object magically becomes an instance of this new class.

The overridden methods are how it does the real work of notifying observers. The logic goes that changes to a key have to go through that key's set method. It overrides that set method so that it can intercept it and post notifications to observers whenever it gets called. (Of course it's possible to make a modification without going through the set method if you modify the instance variable directly. KVO requires that compliant classes must either not do this, or must wrap direct ivar access in manual notification calls.)

It gets trickier though: Apple really doesn't want this machinery to be exposed. In addition to setters, the dynamic subclass also overrides the `-class` method to lie to you and return the original class! If you don't look too closely, the KVO-mutated objects look just like their non-observed counterparts.

Digging Deeper

Enough talk, let's actually see how all of this works. I wrote a program that illustrates the principles behind KVO. Because the dynamic KVO subclass tries to hide its own existence, I mainly use Objective-C runtime calls to get the information we're looking for.

Here's the program: